The Meta in Meta-object Architectures*

Marcus Denker[1] Mathieu Suen[2] Stéphane Ducasse[2]

[1]Software Composition Group
University of Bern – Switzerland

[2]ADAM, INRIA Nord Europe – LIFL –
CNRS UMR 8022 – France

Abstract. Behavioral reflection is crucial to support for example functional upgrades, on-the-fly debugging, or monitoring critical applications. However the use of reflective features can lead to severe problems due to infinite meta-call recursion even in simple cases. This is especially a problem when reflecting on core language features since there is a high chance that such features are used to implement the reflective behavior itself. In this paper we analyze the problem of infinite meta-object call recursion and solve it by providing a first class representation of meta-level execution: at any point in the execution of a system it can be determined if we are operating on a meta-level or base level so that we can prevent infinite recursion. We present how meta-level execution can be represented by a *meta-context* and how reflection becomes *context-aware*. Our solution makes it possible to freely apply behavioral reflection even on system classes: the meta-context brings stability to behavioral reflection. We validate the concept with a robust implementation and we present benchmarks.

1 Introduction

Reflection, a feature common to many modern programming languages, is the ability to query and change a system at runtime [20]. Reflection is a very desirable feature, particularly suited for the long-living and highly dynamic systems of today.

The original model of reflection as defined by Smith [25] is based on meta-level interpretation. The program is interpreted by an interpreter, such interpreter is interpreted by a metainterpreter leading to a tower of interpreters each defining the semantics of the program (the interpreter) it interprets.

However, a tower of interpreters is too slow in practice. To enable reflection in mainstream languages such as CLOS [20], Smalltalk [13, 23, 24] or Java [30], the tower of interpreters is replaced with a reflective architecture [22] where meta-objects control the different aspects of reflection offered by the language. Meta-objects define the new or modified behavior and describe where this new behavior is active. For example, in systems that use metaclasses like CLOS [20], Neoclasstalk [5], or MetaClassTalk [4], the metaclass of a class defines both the new behavior and which classes are affected by the new behavior. Recently,

* R.F. Paige and B. Meyer (Eds.): TOOLS EUROPE 2008, LNBIP 11, pp. 218-237, 2008.

more elaborated schemes have been proposed (*e.g., partial behavioral reflection* [24, 30]) that provide a more flexible and fine-grained way to specify both the location been reflected and the meta-object invoked.

In general, meta-objects provide the implementation of new behavior which is called at certain defined places from the base system. It is important to note that meta-objects are not special objects, the execution of code as part of a meta-object is not different to any execution occuring in the base level application. As both base and metacomputation are handled the same, we are free to call any part of the base-system in the meta-level code.

As a matter of fact, this means that meta-level code can actually trigger again the execution of meta-level functionality. There is nothing to prevent the meta-level code to request the same code to be executed again, leading to an endless loop resulting in a system crash. This is especially a problem when reflecting on core language features (*e.g.,* the implementation of Arrays or Numbers) since the chances are high that such features are used to implement reflective features themselves. These cases of spurious endless recursion of meta-object calls have been noted in the past in the context of CLOS [7].

The ability to reflect on system classes is especially important when using reflection for dynamic analysis. A tracing tool that is realized with reflection should be able to trace a complete run of the system, not only the application code. In addition to the problem of recursion, such a tracer has the problem of recording the execution of trace code itself.

If we go back to the infinite tower (the origin of meta-level architectures) we can see that here these problems do not exist by construction: *going meta* means jumping up to another interpreter. A reflective function is always specific to one interpreter. As a function that is reflective at a meta-level I_n is not necessarily reflective in I_{n+1}, the problem of infinite recursion does not happen.

An important question then is the difference between the meta-object and interpreter/infinite tower approach. The *metaness* in the case of the tower is defined by the interpreter used. The interpreter forms a context that defines if we are executing at the base level or at the meta-level. Calling reflective functionality (so called reification) is always specific to one interpreter. The meta-object approach now in contrast is lacking any mechanism to specify this contextual information: when executing a meta-level program, in a meta-object based reflective system, we lack the information that this program is executing at the meta-level. In addition, all reifications are globally active: we can not define to only trigger meta-object activation when executing base level code. The research question is then how can we incorporate the infinite tower property of explicitly representing the execution context into meta-object based architectures.

Our solution to this problem is to extend meta-object based reflective systems with a first class notion of meta-level execution and the possibility to query at any point in the execution whether we are executing at the meta-level or not. To model meta-level execution we propose the notion of a first class *context* and *context-aware reifications*.

This paper is organized as follows. First we present a simple example to illustrate our problem and elaborate on the system used for evaluating the solution. Section 3 discusses the problem in detail, the next section then provides an overview of *context* and *contextual reflection* which can solve the presented problem (Section 4). We present an implementation in Section 5 followed by an evaluation (Section 6). After an overview of related work (Section 7) we conclude with a discussion of future work in Section 8

2 Context and Example

For the rest of the paper, we discuss the problem of meta-object call recursion in the context of Reflectivity, a reflective library implemented in Squeak [19]. We use Reflectivity to validate our solution. The ideas we discuss in this paper are the outcome of a larger project whose goal is to provide better control of reflection. Thus we started with a model of behavioral reflection that already allows for fine-grained spatial and temporal control of reification [24, 30]. Nevertheless it should be noted that the problem presented in this paper represents a universal problem: we are not just fixing a bug in our Reflectivity framework.

We first provide a short overview of partial behavioral reflection and discuss how to implement a simple example, which we use in the rest of the paper.

2.1 Partial Behavioral Reflection

With Reflex, Tanter introduced *partial behavioral reflection* [30] in the context of Java, the model was later applied to dynamic languages [24]. Here meta-objects are associated not per object (as in 3-KRS [22]) or per metaclass (as in CLOS [20]), but per instruction. The core idea of the Reflex model is to bind meta-objects to operations using a *link* (see Figure 1). One can think about the link as the jump to the meta-level reified as an object. A link thus conceptually invokes messages on a meta-object at occurrences of selected operations.

Link attributes enable further control of the exact message sent to the meta-object. For example, we can control if the meta-object is supposed to be invoked before, after or instead of the original operation, an *activation condition* link attribute controls if the link is really invoked.

For our experiment, we use an implementation of partial behavioral reflection for Smalltalk that uses an abstract syntax tree for selecting which instructions to reflected on. Before execution, the AST is compiled on demand to a low-level representation that is executable, in our case to byte-codes executable by the Squeak virtual machine. More in-depth information about this system and its implementation can be found in the paper on sub-method reflection [10].

Annotating a node of the AST with a link thus results in code that when executed will call the specified meta-object.

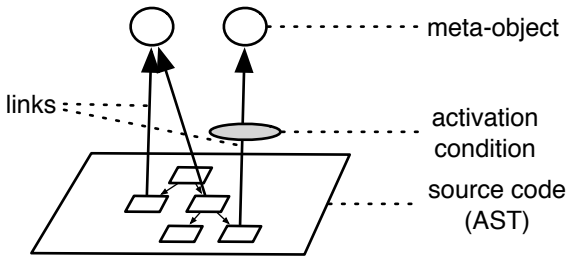

Fig. 1. Partial Behavioral Reflection

2.2 A Simple Example

The problem of meta-object call recursion is a known problem [7]. To show that it is relevant in practice, we have decided to keep our example as simple as possible. In Section 6.2 we discuss as more complex scenario how our solution is useful for dynamic analysis in general.

Imagine that we want to trace system activity: we want the system to beep when it executes a certain method. This audio based debugging is an interesting technique to determine if a certain piece of code is executed or not. In Squeak, there is a class Beeper that provides all the beeping functionality. When calling beep, the beep sound is played via the audio subsystem. The following example shows how to create a link that will invoke the message beep on the Beeper class.

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep
```

Now we can install this beepLink on a method that is part of the system libraries. We take on purpose the method add: in OrderedCollection, a central class part of the collection libraries that is heavily used by the system. To set the link, we invoke the method link: on the AST-Node that stands for the whole method:

```
(OrderedCollection>>#add:) methodNode link: beepLink.
```

As result, a sound should be emited each time the method OrderedCollection»#add: is called. But as soon as we install this link, the system freezes. This clearly was not the intended outcome.

3 Infinite Meta-object Call Recursion

Let's analyze the cause for the problem presented above. After a discussion of some ad-hoc solutions, we show that the problem is caused by a missing model for the concept of the *meta-level execution*.

The Problem. To ensure that the problem is not caused by our framework, we modify the example to call a different method at the meta-object, the method

beepPrimitive which directly executes functionality defined in the virtual machine.

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beepPrimitive.
```

When installing this link, we can see that it works as expected: we can hear a beep for all calls to the add: method, for example when typing characters in the Squeak IDE.

The problem thus lies in the code executed by the meta-object. The Squeak sound subsystem uses the method add: of OrderedCollection at some place to emit the beep sound. Thus, we call the same method add: from the meta-object that triggered the call to the meta in the first place. Therefore we end up calling the meta again and again as shown in Figure 2. This is clearly not a suitable semantics for behavioral reflection: it should be possible to use behavioral reflection even on system libraries that *are used to implement the meta object functionality themselves.*

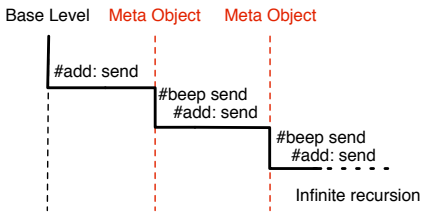

Fig. 2. Infinite recursive meta-object call

We present now two ad-hoc solutions.

Code Duplication. As the problem is caused by calling base level code from the meta-object, one solution would be to never call base level code from the meta-object, but instead provide a renamed copy for all classes and use these from the meta-level. Duplicating the complete system has, of course, all the standard problems of code duplication: space is wasted. In addition, the copy can easily become out of sync with the original. The problems could be minimized by just copying those methods that are really needed. In practice, it is not easy to identify these methods, especially in dynamic languages. In addition this would cause changes in the reflective layer to become fragile because any change would require the programmer to update the copied version of the base level. This is clearly not a good solution.

Adding Special Tests. Another solution could be to add special code to check if a recursion happens. The problem with this solution is that it is ad-hoc, the codebase becomes cluttered with checking instructions. It thus just patches the symptoms, the recursive call, and does not address the real problem.

The Real Problem: Modeling Meta-level Execution. The ad-hoc solutions are not satisfactory. The real problem exemplified by a recursive meta-call is that when going from infinite towers to meta-objects, the awareness that an execution occurs at the meta-level has been lost. Normally activation of a meta-object means a jump to the meta-level. The problem now is that this meta-level does not really exist in meta-object-based architectures: there is no way to query the system to know whether we are at the meta-level or not.

It should be noted again that the problem we have seen is not specific to a particular behavioral reflection framework. We observe the same problem when applying MethodWrappers [6] to system classes. Method wrappers wrap a method with before/after behavior. MethodWrappers are reflectively implemented in Smalltalk and thus use lots of system library code during the execution of the wrapped methods. The same problem was identified for CLOS [7] and is thus present in other meta-class based systems like for example MetaClassTalk [4].

4 Solution: A Metacontext

We have seen that the real cause for the problem of endless recursion lies in the absence of a model for *meta-level execution*: the fact that the system is executing meta-level code does not have a representation.

4.1 Modeling Context

At any point in the execution of some piece of code we should be able to query whether we are executing at the meta or at the base level. Such a property can be nicely modeled with the concept of *context*: the *meta-level* is a context that is active or inactive.

Such a context thus provides control-flow information: we want to know if we are in the control-flow of a call to a meta-object. But in addition, the context actually provides a reification: we have an object representing a specific control-flow, which in our case represents meta-level execution.

With a way to model meta-level execution, it is possible to solve the problem of recursive meta-object calls. A call to the meta-object can be scoped towards the base level: a meta-call should only occur when we are executing at the base level. If we are already at the meta-level, the calls should be ignored. This way, meta-object calls are only triggered by the base level computation, not the meta-level computation itself, thereby eliminating the recursion.

We will first describe a simplified model that only provides two levels of execution (base and meta) and does not allow any metameta-objects to be defined that are activated only from a meta-object execution. We describe later how to extend our model to support calls to these metameta-objects.

The Metacontext. To model the meta-level, we introduce a special *context*, the MetaContext. This context is inactive for a normal execution of a program. MetaContext will be activated when we enter in the meta-level computation and

deactivate when we leave it (Figure 3). The meta-context thus models *meta-level execution*.

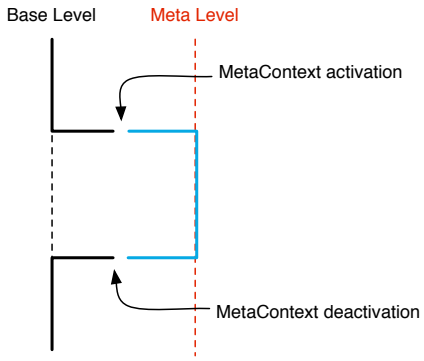

Fig. 3. The MetaContext activation

A simple model with just one meta-context is enough to distinguish the meta from the base level. We will see later that it makes sense to extend the meta-context to a possibly infinite tower of meta-contexts in Section 4.3.

Controlling meta-object activation. Just having a way to model meta-level execution via the meta-context is not enough to solve the problem of recursion, it is just the prerequisite to be able to detect it. We need to make sure that a call to the meta-object does not occur again if we are already executing at the meta-level. Thus, the call to the meta-level needs to be guarded so it is not executed if the execution is already occurring at the meta-level. In the context of behavioral partial reflection (*i.e.,* in the link-meta-object model that we used to show the problem), this means that the links are parameterized by the contexts in which they are active or not-active.

4.2 The Problem Revisited

With both the meta-context and the contextual controlled meta-object calls, we now can return to our example and see how our technique solves the problem of recursion. In our example, we defined the Beeper as a meta-object to be called when executing the add: method of OrderedCollection. The following steps occur (see Figure 4):

1. The add: method is executed from a base level program.
2. A call to the meta-object Beeper is requested:
   - We first check if we are at the meta-level. As we are not, we continue with the call.
   - We enable the MetaContext.
   - We call the meta-object.

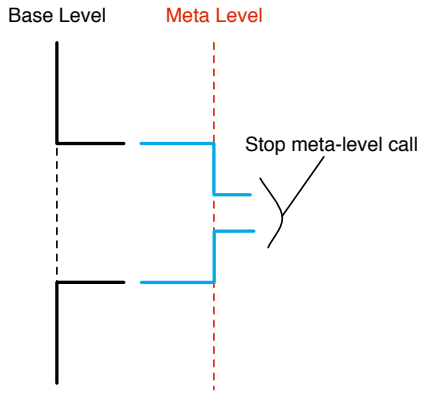

Fig. 4. Stopping infinite meta-call recursion

3. Meta-object executes the `beep` method.
4. Meta-object calls the method `add:` method again
5. A Call to the meta-object is requested
   – We first check if we are at the meta-level. As we are executing meta-level code, the call is aborted.
6. Meta-object execution continues until it is finished.
7. On return to the base level, we deactivate the MetaContext.

Thus the recursive meta-call is aborted and the danger of recursion is eliminated. The model we described up to now with just one MetaContext is thus enough to solve the problem, but it not complete: it does not, for example, allow any calls to metameta-objects while already executing at the meta-level, which would make it impossible to observe or reason about metabehavior. In the next section, we therefore extend the model.

4.3 The Contextual Tower

As with the tower of interpreters, we can generalize the meta-context to form an infinite tower of meta-contexts. With the infinite tower of reflective interpreters, a reification is always bound to a specific interpreter. Normally, a jump from the base to the meta level means executing a reflective function that is defined as part of the interpreter I_1. But it is possible to define a reflective function one level up: this then is only triggered by the interpreter I_2 that interprets I_1, thus allowing to reflect on the interpreter I_1 itself. Figure 5 shows the reflective tower as visualized in the work of Smith [25].

Transposed to our contextual model, it follows that having just one context (the meta-context) is not enough. We need more contexts for meta-meta, meta-3 and so on. If we have this *contextual tower*, we can for example define a meta-meta object that is only called when we are executing at meta-1. Meta-object calls need thus not only be defined to be active for the base level, but they can

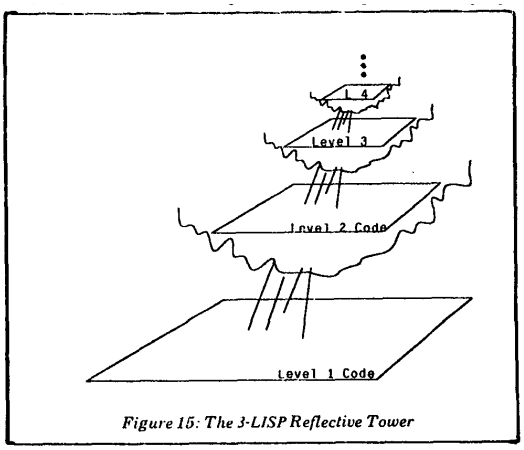

Fig. 5. The 3-Lisp reflective tower from [25]

optionally defined to be active for any of the meta-levels. This allows us to define meta*objects that reason about the system executing at any level, similar to the endless tower of interpreters.

As with all infinite structures, the most important question is how to realize it in practice. For the case of the infinite meta-context tower, there is an easy solution: contexts are objects, they can have state. We can parameterize the meta-context object with a number describing the meta-level that it encodes. Shifting to the meta-level means shifting from a context n to a context $n + 1$.

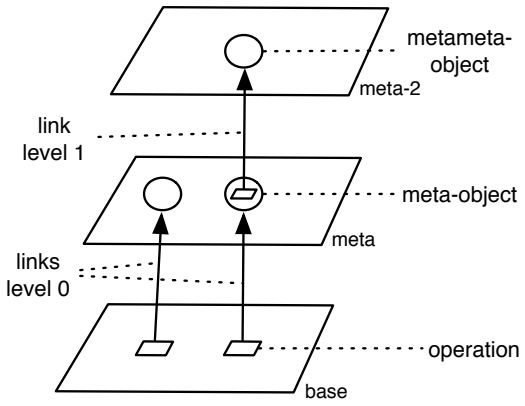

Fig. 6. The Contextual Tower up to the second level

Figure 6 shows an example. We have three contexts: the base level, the meta-level and meta-2. We see two links that are active at the base level. When called,

they activate the meta-context with level 1. Then in the code of the meta-object (either the code of the meta-object itself or any library code executeted), we have a third link. This link is defined to be active only on meta-level 1, thus it will on execution enable meta-2. We show an example of such a level 1 link in Section 6.2.

An interesting and very nice property of the parameterized context is that the meta-contexts are only created on demand, they do not exist as an endless tower. This means, we have a form of partial reflection for providing a potential endless tower. If a meta-level is not needed, it does not cost anything.

4.4 The MetaContext Revised

The simple version of our idea with just one MetaContext is not enough to allow us to encode the Contextual Tower. We need a slightly modified model of MetaContext where the MetaContext is parameterized by the *level*. Such a parameterized MetaContext is not simply active or inactive, it is active for the level that it currently is set to. Thus when querying such a MetaContext, we give as a parameter a number denoting a meta-level. We will in the next section see how to realize such a context in practice.

5 Implementation

Now that we described the solution in general, we present an implementation of that model for Reflectivity, our reflection framework. We first show how the context is implemented and then discuss contextual links.

5.1 Implementation of MetaContext

The MetaContext is a class that has one instance per thread (a thread-specific singleton). The instances are created on demand and are stored per thread. As threads are objects in Squeak, we extended them to be able to store additional state directly in an associated dictionary. This mechanism is then used to store the MetaContext instance.

Querying context. The MetaContext needs to model the meta-level. For that, it has one instance variable named level. We can increase the level by calling shiftLevelUp or decrease by shiftLevelDown. To test if the MetaContext is active for a certain level n we can call isActive: with a parameter denoting the level:

```
MetaLevel current isActive: 0
```

Sending current to the class MetaContext will retrieve the MetaContext singleton from the current process. If there is none yet, it will lazily create a MetaContext with the level set to 0. Thus for a normal base level execution of code the expression above will return true.

Executing code in the MetaContext. To change the meta-level that code is executed at, we provide a way to run a block (an anonymous higher order function) one meta-level higher than the code outside the block. For example, to execute at meta-1, evaluate:

```
[ ... code executing on meta-1 ... ] valueWithMetaContext
```

If code is already executing at meta-1, calling valueWithMetaContext again will execute at meta-2:

```
[[ ... code executing on meta-2 ... ] valueWithMetaContext] valueWithMetaContext
```

The method valueWithMetaContext is implemented on the BlockClosure object, it will first shift the level of the current MetaContext up, then the block is executed. At the end the level of the context is shifted down to the previous value. We make sure that the downshift happens even in case of abnormal termination by evaluating the block using the exception handling mechanisms of Smalltalk:

```
valueWithMetaContext
   MetaContext current shiftLevelUp.
   ^ self ctxtEnsure: [MetaContext current shiftLevelDown]
```

To make sure that the execution of the context handling code itself does not result in endless loops, we do not call any code from the sytem libraries in this method. Instead, we carefully copy all methods executed by the context setup code. The copied methods reside in the classes of the system library, but they are prefixed with *ctxt* and edited to call only prefixed methods. One example is the call to ctxtEnsure: seen in the method above.

Concurrent meta-objects. As the MetaContext is represented by a thread-specific singleton, forking a new thread from the meta-level would mean that this thread has its own MetaContext object associated which is initialized to be at level 0. We have solved this problem by changing the implementation of threads to actually copy the meta-level information to the newly created thread. A thread created at the meta-level thus continues to run at the meta-level:

```
[ [ self assert: (MetaContext current isActive: 1) ]  fork ] valueWithMetaContext
```

As the context information is not shared with the parent thread, the meta-level of the new thread is independed, it can continue to run at the meta-level even when the parent thread already returned to the base level.

5.2 Realizing Contextual Links

Now that we have a suitable way to represent the meta context, we need to make the links and the code that is generated to call them *context-aware*. For that, we need to solve three problems:

1. the link needs to be defined to be specific to a certain meta level.

2. link activation should occur only when code is executing on the right meta-level.
3. link activation should increase the meta-level.

The first problem is solved by a simple extension of the Link class, whereas the other two are concerned with the code that our system generates for link activation. We will now show the required changes in detail.

Meta-level Specific Links. To allow the programmer to specify that a link is specific to a certain meta-level, we extend the link with a parameter called *level*. If *level* is not set, the link is globally active over all links (the standard behavior). The level can be set to any integer to define a link to only be active on that specific meta level. For our example, a link that is active only when executing base level code looks like this:

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
beepLink level: 0.
```

Context-Aware Link Activation. To jump up one meta-level on link activation, we make sure that the code generated for a link is wrapped in a call to valueWithMetaContext. The resulting code will look like this:

```
[ ... code of the link  ... ] valueWithMetaContext
```

In addition to that, we need to make sure that the link is only called when we are on the correct meta level. This is done by checking if the current MetaContext is executing on the same level as the level that the link is defined to be active. Only if this is true, we activate the link and call the meta-object. The code we need to generate looks like this:

```
(MetaLevel current isActive: link level)  ifTrue: [
    [ ... code of the link  ... ] valueWithMetaContext
].
```

We will not go into the detail of how exactly the code is generated. The code can be found in the class GPTransformer of the Reflectivity distribution[2].

6 Evaluation and Benchmarks

We first show that our solution solves the recursion problem, then we discuss how meta-context is useful for dynamic analysis. We describe *dynamic meta-level analysis* and realize an example. We present benchmarks to show the practicability of our approach.

[2] `http://www.iam.unibe.ch/~scg/Research/Reflectivity`

6.1 The Problem is Solved

We show that we can really solve the practical problem. For that, we define the link that activates the Beeper to be specific to level 0:

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
beepLink level: 0.
```

Now we can install the link:

```
(OrderedCollection>>#add:) methodNode link: beepLink.
```

As soon as the link is installed, the next call to the add: method will trigger code-generation for that method. The code-generator will take the link into account and generate code as described earlier. Thus the recursive call to the add: method will not occur. We will thus hear a beep for every call to the add: method from the base level only.

6.2 Benefits for Dynamic Analysis

Our initial reason for modeling meta-level execution was to solve the problem of meta-object call recursion, and thus making reflection easier to use. But the solution we presented, modeling meta-level execution with the help of a meta-context, is useful far beyond just solving the problem of recursion. In this section, we will discuss what it means for dynamic analysis. We show how it allows the programmer to analyze the code executed by meta-objects by enabling *dynamic meta-level analysis*.

Dynamic Analysis. One of the applications for behavioral reflection is dynamic analysis. For example, with reflection, it is fairly easy to introduce tracers or profilers into a running program [12] that normally require changes at the level of the virtual machine. One problem, though, with using reflection to introduce analysis code is that it is not clear which of the recorded events are resulting from the base level program execution and which from the code of the tracer itself. As long as we only trace application code, we can easily restrict the reflective tracer to the code of the application. But as soon as we want a complete system trace, we start to get into problems: recursion can occur easily (the problem we solved earlier), but even after working around recursion, we face another problem: how do we distinguish events that originated from the application from those that only occur due to the code executed by the tracer itself?

With our meta-level execution context, the problem described does not occur at all. The tracer (or any other tool performing dynamic analysis) is actually a meta-level program. A simple tracer would be the meta-object itself. More complex analysis tools would be called from a meta-object reifying the runtime event we are interested in. Thus, the code that performs the dynamic analysis is executing at the meta-level, while the links that trigger it are only active when executing a base level computation. This way we make sure that the infrastructure performing our analysis never affects the trace itself.

Dynamic Meta-level Analysis. An interesting challenge for dynamic analysis is that of analyzing the meta-level execution itself. Meta-level code should be lean and fast to not slow down the base computation more then really necessary. We thus are very interested in both tracing and profiling meta-level execution.

Our explicitly modeled meta-level execution makes this easy: we can define a link to be only active when executing at the meta-level. Therefore, we can install for example a trace-tool that exactly provides a trace of only those methods executed by a meta-object, even though the same methods are are called by the base level at the same time. We can thus as easily restrict the tracer towards meta-level execution as we restrict it to trace base level programs.

For our example, this means that we can use dynamic meta-level analysis to find the place in the sound-subsystem where the recursion problem happens when not using contextual links. In Section 3 we discuss that recursion happens, but we do not know where exactly the recursive call happens.

We define a link that is only active when we are executing code at level 1:

```
loggerLink := GPLink new metaObject: logger;
                    selector: #log:;
                    arguments: #(context);
                    level: 1.
```

This link sends the message log: to the logger. The logger is an instance of class MyLogger:

```
logger := MyLogger new.
```

The method log: records the stack-frame that called the method where the link is installed on:

```
MyLogger>>log: aContext
        contexts add: aContext home sender copy
```

We install both a link calling the Beeper that is specific to level 0 and our link that is specific to level 1 on the method add:.

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
beepLink level: 0.

(OrderedCollection>>#add:) methodNode link: beepLink.
(OrderedCollection>>#add:) reflectiveMethod methodNode link: loggerLink.
```

We can now inspect the logger object and see that it is recording the execution of SoundPlayer class»startPlayingImmediately: for every beep. Looking at this method, we find the code ActiveSounds add: aSound., which is the one spot in the sound system that calls the method add: of OrderedCollection. Thus we found with the help of dynamic meta-level analysis the exact call that causes the recursion problem as shown in Section 3.

6.3 Benchmarks

To assess if the system as presented is practically usable, we have carried out some benchmarks. Without the additional code for context activation, there is no overhead at all for calling a meta-object besides the call itself. The link specifies the meta-object and which method to call. The system generates from that information code that just calls the meta as specified. The problem now for analyzing the new context-enabled system is that the context code will, compared to the standard link activation, take a lot of time. In practice, though, meta-objects are usually there to do something: there is always code executed at the meta-level. So to make a practical comparison of the additional percentage of slowdown introduced by the context code, we need to compare the context-setup code not only to the link activation of an empty meta-object, but to a meta-object that actually executes code.

We will benchmark the slowdown of the context handling for the execution of different meta-objects. The variation between the meta-objects is the number of sends done at the meta-level. For the benchmark, we create a class Base with just one empty method bench. To play the role of a meta-object, we create a class Meta with one method in which we call an empty method in a loop. This method thus simulates a meta-object doing some work. We can easily change the amount of work done by changing the loop. To know how this simple benchmark compares to real code executed, we added a meta-object calling the Beeper and one converting a float to a string.

We install a link on the method in Base to call the method in Meta. We call the base method now in a loop and measure the time:

```
[100000 timesRepeat: [Base new bench ]] timeToRun
```

Table 1 shows the result when comparing both the original Geppetto and the context-enabled Geppetto for different meta-objects[3]:

As expected, the slowdown in the case of an empty meta-object is substantial. But as soon as the meta-object itself executes code, the overhead starts to be acceptable. For calling the Beeper (from our running example), we have found an overhead of 63%. We are down to 17% on when executing 500 empty methods, and at 6.4% on 1000 methods.

It should be noted that this does not mean that the overhead observed for meta-object calls translate directly into a slowdown of a program using reflection. In a real program, the overall slowdown depends on how often meta-objects are called and how much time the program spends at the base level and meta-level compared to switching meta-levels.

[3] The benchmark was run on an Apple MacBook Pro, 2.4Ghz Intel Core 2 Duo with 2GB RAM on Squeak Version 3.9

meta-object	context (msecs)	standard (msecs)	slowdown
0 message sends	614	34	1'705.88%
10 message sends	723	165	338.18%
50 message sends	1040	470	121.28%
Beeper	1543	942	63.80%
100 message sends	1406	856	64.25%
200 message sends	2236	1621	37.94%
1234.345 printString	2534	1920	31.98%
500 message sends	4580	3907	17.23%
1000 message sends	8543	8029	6.40%

Table 1. Slowdown of meta-object calls with context

7 Related Work

Meta-object Architectures. There are many examples for meta-object based reflective systems. Examples are 3-KRS [22], the CLOS MOP [20] and other metaclass based systems like Neoclasstalk [5], or MetaClassTalk [4]. A more recent example is partial behavioral reflection for Java [30] and Smalltalk [24]. None of these systems provide a representation for meta-level execution.

One could even argue that all these systems are not really reflective (as they can not be used to reflect on the system itself) and it is debatable if the meta-objects in these systems are really *meta*. It has already noted by Ferber [14] that metaclasses are not meta in the computational sense, even though they are meta in the structural sense. For any computationally reflective system, we need to be able to decide if a computation is executing at the meta-level or not. Without this, there is no reason to talk about *meta-objects* at all.

Aspect Oriented Programming. There is an ongoing debate about the relationship of aspect oriented rogramming (AOP) [21] and reflection. Proponents of reflection claim that AOP is just a use-case for behavioral reflection, a pattern that can be easily implemented. The AOP community, though, would claim that reflection in turn is just the case of an aspect system where the domain is the language model itself, with the core-concepts (e.g. message sending) factored nicely into aspects and thus easily modifiable.

The ideas from this paper can contribute to this discussion. In AOP, a pointcut is globally visible: it matches even in the advice code itself by default. Conversely, we can say that the problem we noted in this paper is made an explicit and well known property in AOP, whereas it is a bug to be fixed in meta-object based reflection.

We claim that exactly here lies the difference between reflection and AOP: reflection needs the distinction between the base and the meta. Aspects, however, are a pure base level abstraction. Invoking an advice does not constitute a metacomputation, a level shift does not occur.

Contrary the notion that aspects are pure base level constructs, *Stratified Aspects* [3] define an extension that identify spurious recursion to be a problem and present the concept of metaadvice and metaaspects as a solution. This idea has some similarities to contextual reifications discussed in this paper, with the exception that meta-level execution is not modeled explicitly.

Context-Oriented Programming. ContextL [8, 18] is a language to support *Context-Oriented Programming* (COP). The language provides a notion of *layers*, which package context-dependent behavioral variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. COP has no first-class notion of *context*, it is implicitly defined by the layers that are activated. The topic of reflection has been discussed for COP [9]. But the paper looks at reflective layer activation, not a reflective model of context nor the use of context to structure or control reflection itself.

Context-aware Aspects. The concept of context has seen some use in AOP [29]. As context specific behavior tends to crosscut base programs, it can advantageously be implemented as aspects. This leads to the notion of context-aware aspects, i.e., aspects whose behavior depends on context. The work has been continued in the direction of supporting reasoning on contexts and context history on the level of the pointcuts [16, 17].

Deployment strategies [28] provide full control over dynamic aspect deployment both related to the call stack and to created objects and functions. AspectBoxes [2] are another example where aspects are controlled via a form of a context, in this case defined by the classbox model [1]. A good overview and discussion on the many mechanisms for dynamically scoping crosscutting in general is described in the work of Tanter [27].

The MetaHelix. The problem of unwanted meta-level call recursion has been mentioned by Chiba and Kiczales [7]. The problem discussed is first the structural problem that *e.g.,* fields added by reflection to implement changed behavior show through to any user of introspection. The other problem mentioned is recursion, as any use of changed behavior can trigger a reification again. As a solution, the authors present the *MetaHelix*. All meta-objects have a field implemented-by that points to a version of the code that is not reflectively changed.

This approach is both more general and restrictive than our context based solution. It is more general, as it tries to solve the problem of the visibility of structural change. And it is more restrictive, as it does not model meta-level execution. The programmer has to call the right code explicitly, thus it can be seen as a controlled way to support the code copying solution presented in Section 3. The problem of structural changes is a very interesting one, as future work we plan to apply the ideas of the meta-context to structural reflection.

Subjective Programming. Us [26] is a system based on Self that supports subject-oriented programming [15]. Message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContexL. The paper discusses the usefulness of subjectivity for controlling the access to reflection APIs, it does not go as far as using subjectivity for controlling behavioral reflection.

8 Conclusion and Future Work

In this paper we have analyzed the problem of the missing representation of meta-level execution in meta-object architectures. We have shown that the problem of infinite meta-object call recursion can be solved by introducing a representation for meta-level execution. We proposed to model the execution at the meta-level as a first class context and presented an implementation. Benchmarks show that the implementation can be realized in a practical manner.

For now, we have used the concept of *context* just to make the meta computation distinguishable from the base computation. We plan to extend the notion of contextual control of reification to other kinds of contexts then the meta-context.

We have experimented in the past with the idea of a first class model of change for programming languages [11]. We will explore the idea of context for structural reflection to model change. Virtual machine support for meta-contexts is interesting for two reasons. First, we hope to be able to improve performance by realizing all context setup code in the virtual machine. Second, as we explained in Section 5, the setup code executed when dealing with contexts has to be managed specially: we provide copies of all that code. We plan to move all this special code into the virtual machine.

An interesting question is how a context-aware reflective language kernel would look like and what the consequences for the language runtime and especially the reflective model would be. We plan to explore such a new reflective language kernel in the future.

Acknowledgments. We thank Eric Tanter, Robert Hirschfeld, Orla Greevy, Oscar Nierstrasz, Martin von Löwis, Toon Verwaest, Adrian Kuhn, Adrian Lienhard and Lukas Renggli for discussing various aspects of the concepts presented in this paper. We acknowledge the financial support of the Swiss National Science Foundation for the project “Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and COOK (JC05 42872) funded by the french Agence Nationale de la Recherche.

References

1. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.
2. Alexandre Bergel, Robert Hirschfeld, Siobhàn Clarke, and Pascal Costanza. Aspectboxes — controlling the visibility of aspects. In Markus Helfert Joaquim Filipe, Boris Shiskov, editor, *In Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2006)*, pages 29–38, September 2006.
3. Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, editors, *GI-Edition Lecture Notes in Informatics "NODe 2006 GSEM 2006"*, volume P-88, pages 49–64. Gesellschaft für Informatik, Bonner Köllen Verlag, 2006.

4. Noury Bouraqadi. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects (A Smalltalk MOP for the Study of Metaclass Composition and Compatibility. Application to Aspect-Oriented Programming - In French).* Thèse de doctorat, Université de Nantes, Nantes, France, jul 1999.
5. Noury Bouraqadi, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *Proceedings OOPSLA '98*, pages 84–96, 1998.
6. John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.
7. Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.
8. Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.
9. Pascal Costanza and Robert Hirschfeld. Reflective layer activation in ContextL. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1280–1285, New York, NY, USA, 2007. ACM Press.
10. Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, October 2007.
11. Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.
12. Marcus Denker, Orla Greevy, and Michele Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.
13. Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.
14. Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.
15. William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.
16. Charlotte Herzeel, Kris Gybels, and Pascal Costanza. A temporal logic language for context awareness in pointcuts. In *Proceeding of the Workshop on Revival of Dynamic Languages*, 2006.
17. Charlotte Herzeel, Kris Gybels, Pascal Costanza, and Theo D'Hondt. Modularizing crosscuts in an e-commerce application in lisp using halo. In *Proceeding of the International Lisp Conference (ILC) 2007*, 2007.
18. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.
19. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

20. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
21. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.
22. Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.
23. Fred Rivard. Pour un lien d'instanciation dynamique dans les langages à classes. In *JFLA96*. INRIA — collection didactique, January 1996.
24. David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.
25. Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.
26. Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.
27. Éric Tanter. On dynamically-scoped crosscutting mechanisms. *ACM SIGPLAN Notices*, 42(2):27–33, February 2007.
28. Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, April 2008. ACM Press. To appear.
29. Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *LNCS*, pages 227–242, Vienna, Austria, March 2006.
30. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.